UNITED STATES DESIGN PATENT APPLICATION

FOR

VECTORING AN INTERRUPT OR EXCEPTION UPON RESUMING OPERATION OF A VIRTUAL MACHINE

Inventors:

STEPHEN M. BENNETT

ANDREW V. ANDERSON

STALINSELVARAJ JEYASINGH

ALAIN KAGI

GILBERT NEIGER

RICHARD UHLIG

MICHAEL KOZUCH

LAWRENCE SMITH

SCOTT RODGERS

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, CA 90025-1026

(408) 720-8300

EXPRESS MAIL CERTIFICATE OF MAILING

"Express Mail" mailing label number EV336588670US
Date of Deposit September 15, 2003
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450

Michelle Begay
(Typed or printed name of person mailing paper or fee)

Michelle Begay
(Signature of person mailing paper or fee)

VECTORING AN INTERRUPT OR EXCEPTION UPON RESUMING OPERATION OF A VIRTUAL MACHINE

Field

[0001] Embodiments of the invention relate generally to virtual machines, and more specifically to handling faults in a virtual machine environment.

Background

[0002] A conventional virtual-machine monitor (VMM) typically runs on a computer and presents to other software the abstraction of one or more virtual machines. Each virtual machine may function as a self-contained platform, running its own "guest operating system" (i.e., an operating system (OS) hosted by the VMM) and other software, collectively referred to as guest software. The guest software expects to operate as if it were running on a dedicated computer rather than a virtual machine. That is, the guest software expects to control various events and have access to hardware resources. The hardware resources may include processor-resident resources (e.g., control registers), resources that reside in memory (e.g., descriptor tables) and resources that reside on the underlying hardware platform (e.g., input-output devices). The events may include internal interrupts, external interrupts, exceptions, platform events (e.g., initialization (INIT) or system management interrupts (SMIs)), and the like.

[0003] In a virtual-machine environment, the VMM should be able to have ultimate control over the events and hardware resources as described in the previous paragraph to provide proper operation of guest software running on the virtual machines and for protection from and among guest software running on the virtual machines. To achieve this, the VMM typically receives control when guest software accesses a protected resource or when other events (such as interrupts or exceptions) occur. For example, when an operation in a virtual machine supported by the VMM causes a system device to generate an interrupt, the currently running virtual machine is interrupted and control of the processor is passed to the VMM. The VMM then receives the interrupt, and handles the interrupt itself or invokes an appropriate virtual machine and delivers the interrupt to that virtual machine.

Brief Description of the Drawings

[0004] The invention may be best understood by referring to the following description and accompanying drawings that are used to illustrates embodiments of the invention. In the drawings:

[0005] Figure 1 illustrates one embodiment of a virtual-machine environment, in which some embodiments of the present invention may operate;

[0006] Figure 2 is a flow diagram of one embodiment of a process for handling faults in a virtual machine environment;

[0007] Figure 3 illustrates an exemplary format of a VMCS field that stores fault identifying information;

[0008] Figure 4 is a flow diagram of one embodiment of a process for handling a fault in a virtual-machine environment using fault information provided by a VMM.

Description of Embodiments

[0009] A method and apparatus for handling a fault in a virtual-machine environment using fault information provided by a VMM are described. In the following description, for purposes of explanation, numerous specific details are set forth. It will be apparent, however, to one skilled in the art that embodiments of the invention can be practiced without these specific details.

[0010] Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer system's registers or memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to convey the substance of their work to others skilled in the art most effectively. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, although not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0011] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically

stated otherwise as apparent from the following discussions, it is appreciated that discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or the like, may refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer-system memories or registers or other such information storage, transmission or display devices.

[0012] In the following detailed description of the embodiments, reference is made to the accompanying drawings that show, by way of illustration, specific embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, and electrical changes may be made without departing from the scope of the present invention. Moreover, it is to be understood that the various embodiments of the invention, although different, are not necessarily mutually exclusive. For example, a particular feature, structure, or characteristic described in one embodiment may be included within other embodiments.

[0013] Although the below examples may describe embodiments of the present invention in the context of execution units and logic circuits, other

embodiments of the present invention can be accomplished by way of software. For example, in some embodiments, the present invention may be provided as a computer program product or software which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. In other embodiments, steps of the present invention might be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[0014] Thus, a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, a transmission over the Internet, electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.) or the like.

[0015] Further, a design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as is useful in simulations, the

hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, data representing a hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine-readable medium. An optical or electrical wave modulated or otherwise generated to transmit such information, a memory, or a magnetic or optical storage such as a disc may be the machine readable medium. Any of these mediums may "carry" or "indicate" the design or software information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is performed, a new copy is made. Thus, a communication provider or a network provider may make copies of an article (a carrier wave) embodying techniques of the present invention.

[0016] Figure 1 illustrates a virtual-machine environment 100, in which some embodiments of the present invention may operate. In the virtual-machine environment 100, bare platform hardware 110 comprises a computing platform, which may be capable, for example, of executing a

standard operating system (OS) and/or a virtual-machine monitor (VMM), such as a VMM 112. The VMM 112, though typically implemented in software, may emulate and export a bare machine interface to higher level software. Such higher level software may comprise a standard or real-time OS, may be a highly stripped down operating environment with limited operating system functionality, or may not include traditional OS facilities. Alternatively, for example, the VMM 112 may be run within, or on top of, another VMM. VMMs and their typical features and functionality are well known by those skilled in the art and may be implemented, for example, in software, firmware, hardware or by a combination of various techniques.

[0017] The platform hardware 110 includes a processor 118 and memory 120. Processor 118 can be any type of processor capable of executing software, such as a microprocessor, digital signal processor, microcontroller, or the like. Though only one processor 118 is shown in **Figure 1**, the platform hardware 110 may include one or more such processors.

[0018] Memory 120 can be any type of recordable/non-recordable media (e.g., random access memory (RAM), read only memory (ROM), magnetic disk storage media, optical storage media, flash memory devices, etc.), as well as electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.), any combination of the above devices, or any other type of machine medium readable by processor 118. Memory 120 may store instructions for performing the execution of method embodiments of the present invention.

[0019] The platform hardware 110 can be of a personal computer (PC), mainframe, handheld device, portable computer, set-top box, or any other computing system.

[0020] The VMM 112 presents to other software (i.e., "guest" software) the abstraction of one or more virtual machines (VMs), which may provide the same or different abstractions to the various guests. **Figure 1** shows three VMs, 130, 140 and 150. The guest software running on each VM may include a guest OS such as a guest OS 154, 160 or 170 and various guest software applications 152, 162 and 172.

[0021] The guest OSs 154, 160 and 170 expect to access physical resources (e.g., processor registers, memory and input-output (I/O) devices) within corresponding VMs (e.g., VM 130, 140 and 150) on which the guest OSs are running and to perform other functions. For example, the guest OS expects to have access to all registers, caches, structures, I/O devices, memory and the like, according to the architecture of the processor and platform presented in the VM. The resources that can be accessed by the guest software may either be classified as "privileged" or "non-privileged." For privileged resources, the VMM 112 facilitates functionality desired by guest software while retaining ultimate control over these privileged resources. Non-privileged resources do not need to be controlled by the VMM 112 and can be accessed by guest software.

[0022] Further, each guest OS expects to handle various fault events such as exceptions (e.g., page faults, general protection faults, etc.), interrupts

(e.g., hardware interrupts, software interrupts), and platform events (e.g., initialization (INIT) and system management interrupts (SMIs)). Some of these fault events are "privileged" because they must be handled by the VMM 112 to ensure proper operation of VMs 130 through 150 and for protection from and among guest software.

[0023] When a privileged fault event occurs or guest software attempts to access a privileged resource, control may be transferred to the VMM 112. The transfer of control from guest software to the VMM 112 is referred to herein as a VM exit. After facilitating the resource access or handling the event appropriately, the VMM 112 may return control to guest software. The transfer of control from the VMM 112 to guest software is referred to as a VM entry. In one embodiment, the VMM 112 requests the processor 118 to perform a VM entry by executing a VM entry instruction.

[0024] In one embodiment, the processor 118 controls the operation of the VMs 130, 140 and 150 in accordance with data stored in a virtual machine control structure (VMCS) 126. The VMCS 126 is a structure that may contain state of guest software, state of the VMM 112, execution control information indicating how the VMM 112 whishes to control operation of guest software, information controlling transitions between the VMM 112 and a VM, etc. In one embodiment, the VMCS is stored in memory 120. In some embodiments, multiple VMCS structures are used to support multiple VMs.

[0025] When a privileged fault event occurs, the VMM 112 may handle the fault itself or decide that the fault needs to be handled by an appropriate

VM. If the VMM 112 decides that the fault is to be handled by a VM, the VMM 112 requests the processor 118 to invoke this VM and to deliver the fault to this VM. In one embodiment, the VMM 112 accomplishes this by setting a fault indicator to a delivery value and generating a VM entry request. In one embodiment, the fault indicator is stored in the VMCS 126.

[0026] In one embodiment, the processor 118 includes fault delivery logic 124 that receives the request of the VMM 112 for a VM entry and determines whether the VMM 122 has requested the delivery of a fault to the VM. In one embodiment, the fault delivery logic 124 makes this determination based on the current value of the fault indicator stored in the VMCS 126. If the fault delivery logic 124 determines that the VMM has requested the delivery of the fault to the VM, it delivers the fault to the VM when transitioning control to this VM.

[0027] In one embodiment, delivering of the fault involves searching a redirection structure for an entry associated with the fault being delivered, extracting from this entry a descriptor of the location of a routine designated to handle this fault, and jumping to the beginning of the routine using the descriptor. Routines designated to handle corresponding interrupts, exceptions or any other faults are referred to as handlers. In some instruction set architectures (ISAs), certain faults are associated with error codes that may need to be pushed onto stack (or provided in a hardware register or via other means) prior to jumping to the beginning of the handler.

[0028] During the delivery of a fault, the processor 118 may perform one or more address translations, converting an address from a virtual to physical form. For example, the address of the interrupt table or the address of the associated handler may be a virtual address. The processor may also need to perform various checks during the delivery of a fault. For example, the processor may perform consistency checks such as validation of segmentation registers and access addresses (resulting in limit violation faults, segment-not-present faults, stack faults, etc.), permission level checks that may result in protection faults (e.g., general-protection faults), etc.

[0029] Address translations and checking during fault vectoring may result in a variety of faults, such as page faults, general protection faults, etc. Some faults occurring during the delivery of a current fault may cause a VM exit. For example, if the VMM 112 requires VM exists on page faults to protect and virtualize the physical memory, then a page fault occurring during the delivery of a current fault to the VM will result in a VM exit.

[0030] In one embodiment, the fault delivery logic 124 addresses the above possible occurrences of additional faults by checking whether the delivery of the current fault was successful. If the fault delivery logic 124 determines that the delivery was unsuccessful, it further determines whether a resulting additional fault causes a VM exit. If so, the fault delivery logic 124 generates a VM exit. If not, the fault delivery logic 124 delivers the additional fault to the VM.

[0031] **Figure 2** is a flow diagram of one embodiment of a process 200 for handling faults in a virtual machine environment. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as that run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 200 is performed by fault delivery logic 124 of **Figure 1**.

[0032] Referring to **Figure 2**, process 200 begins with processing logic receiving a request to transition control to a VM from a VMM (processing block 202). In one embodiment, the request to transition control is received via a VM entry instruction executed by the VMM.

[0033] At decision box 204, processing logic determines whether the VMM has requested a delivery of a fault to the VM that is to be invoked. A fault may be an internal interrupt (e.g., software interrupt), an external interrupt (e.g., hardware interrupt), an exception (e.g., page fault), a platform event (e.g., initialization (INIT) or system management interrupts (SMIs)), or any other fault event. In one embodiment, processing logic determines whether the VMM has requested the delivery of a fault by reading the current value of a fault indicator maintained by the VMM. The fault indicator may reside in the VMCS or any other data structure accessible to the VMM and processing logic 200. In one embodiment, when the VMM wants to have a fault delivered to a VM, the VMM sets the fault indicator to the delivery value and then generates a request to transfer control to this VM. If no fault

delivery is needed during a VM entry, the VMM sets the fault indicator to a no-delivery value prior to requesting the transfer of control to the VM. This is discussed below with respect to **Figure 3**.

[0034] If processing logic determines that the VMM has requested a delivery of a fault, processing logic delivers the fault to the VM while transitioning control to the VM (processing block 206). Processing logic then checks whether the delivery of the fault was successful (decision box 208). If so, process 200 ends. If not, processing logic determines whether a resulting additional fault causes a VM exit (decision box 210). If so, processing logic generates a VM exit (processing block 212). If not, processing logic delivers the additional fault to the VM (processing block 214), and, returning to processing block 208, checks whether this additional fault was delivered successfully. If so, process 200 ends. If not, processing logic returns to decision box 210.

[0035] If processing logic determines that the VMM has not requested a delivery of a fault, processing logic transitions control to the VM without performing any fault related operations (processing block 218).

[0036] In one embodiment, when processing logic needs to deliver a fault to a VM, it searches a redirection structure (e.g., the interrupt-descriptor table in the instruction set architecture (ISA) of the Intel® Pentium® 4 (referred to herein as the IA-32 ISA)) for an entry associated with the fault being delivered, extracts from this entry a descriptor of a handler associated with this fault, and jumps to the beginning of the handler using the descriptor. The

interrupt-descriptor table may be searched using fault identifying information such as a fault identifier and a fault type (e.g., external interrupt, internal interrupt, non-maskable interrupt (NMI), exception, etc.). In one embodiment, certain faults (e.g., some exceptions) are associated with error codes that need to be pushed onto stack (or provided in a hardware register or via other means) prior to jumping to the beginning of the handler. In one embodiment, the fault identifying information and associated error code are provided by the VMM using a designated data structure. In one embodiment, the designated data structure is part of the VMCS. **Figure 3** illustrates an exemplary format of a VMCS field that stores fault identifying information. This VMCS field is referred to as a fault information field.

[0037] Referring to **Figure 3**, in one embodiment, the fault information field is a 32-bit field in which the first 8 bits store an identifier of a fault (e.g., an interrupt or exception), the next 2 bits identify the type of the fault (e.g., external interrupt, software interrupt, NMI, exception, etc.), bit 11 indicates whether an error code (if any) associated with this fault is to be provided to a corresponding handler (by pushing onto the stack, stored in a hardware register, etc.), and bit 31 is a fault indicator that specifies whether a fault is to be delivered to a VM as discussed above.

[0038] If bit 11 of the fault information field indicates that an error code is to be provided to the handler, a second VMCS field is accessed to obtain the error code associated with this fault. The second VMCS field is referred to as a fault error code field.

[0039] In one embodiment, when the VMM wants a fault to be delivered to a VM, the VMM stores the fault identifier and the fault type in the fault information field and sets the fault indicator (bit 31) to a delivery value (e.g., bit 31=1). In addition, if the fault is associated with an error code that needs to be provided to the handler, the VMM sets bit 11 to a delivery value (e.g., bit 11=1) and stores the error code value in the fault error code field in the VMCS.

[0040] **Figure 4** is a flow diagram of one embodiment of a process 400 for handling a fault in a virtual-machine environment using fault information provided by a VMM. The process may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, microcode, etc.), software (such as run on a general purpose computer system or a dedicated machine), or a combination of both. In one embodiment, process 400 is performed by fault delivery logic 124 of **Figure 1**.

[0041] Referring to **Figure 4**, process 400 begins with processing logic detecting an execution of a VM entry instruction by the VMM (processing block 402). In response, processing logic accesses a fault indicator bit controlled by the VMM (processing block 403) and determines whether a fault is to be delivered to the VM that is to be invoked (decision box 404). If not, processing logic ignores the remaining fault information and performs the requested VM entry (processing block 406). If so, processing logic obtains fault information from a fault information field in the VMCS (processing block 408) and determines whether an error code associated with this fault is

to be provided to the fault's handler (decision box 410). If so, processing logic obtains the error code from a fault error code field in the VMCS (processing block 412). If not, processing logic proceeds directly to processing block 414.

[0042] At processing block 414, processing logic delivers the fault to the VM while performing the VM entry. Processing logic then checks whether the delivery of the fault was successful (decision box 416). If so, process 400 ends. If not, processing logic determines whether a resulting additional fault causes a VM exit (decision box 418). If so, processing logic generates a VM exit (processing block 420). If not, processing logic delivers the additional fault to the VM (processing block 422), and, returning to processing block 416, checking whether this additional fault was delivered successfully. If so, process 400 ends. If not, processing logic returns to decision box 418.

[0043] Thus, a method and apparatus for handling faults in a virtual machine environment have been described. It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.